segregated list

14-513/18-613, Summer 2020
Malloc Lab: Writing a Dynamic Storage Allocator
Assigned: Mon, Jun 29

This lab requires submitting two versions of your code: one as an initial checkpoint, and the second as your final version. The dates and weights for your course grade are indicated in the following table:

Version	Due Date	Max. Grace Days	Last Date	Weight in Course
Checkpoint	Jul 6	2	Jul 8	4%
Final	Jul 13	2	Jul 15	7%

1 Introduction

In this lab you will be writing a *general purpose* dynamic storage allocator for C programs; that is, your own version of the `malloc`, `free`, `realloc`, and `calloc` functions. You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient, and fast.

In order to get the most out of this lab, we *strongly* encourage you to start early. The total time you spend designing and debugging can easily eclipse the time you spend coding.

Read this writeup **very carefully**. You should peruse the tips and the advice sections before beginning the development at all. Whenever you have trouble understanding the infrastructure behind this lab, consult this writeup to see if the information you need is specified here.

Bugs can be especially pernicious and difficult to track down in an allocator, and you will probably spend a significant amount of time debugging your code. *Buggy code will not get any credit. You cannot afford to get score of 0 for this assignment.*

This lab has been heavily revised from previous versions (really!). Do not rely on advice or information you may find on the Web or from people who have done this lab before. It will most likely be misleading or outright wrong. Be sure to read all of the documentation carefully and especially study the baseline implementation we have provided.

2 Logistics

This is an individual project. You should do this lab on one of the Shark machines.

3 Source Code Instructions

The only file you will be turn in is `mm.c`*, which contains your solution.* The provided code allows you to locally evaluate your implementations. Using the command `make` will generate three *driver* programs: `mdriver`, `mdriver-dbg`, and `mdriver-emulate`. Use these to evaluate the correctness and performance of your implementations. The driver program used to autograde your code is `driver.pl`, which computes your final score.

To test your code for the checkpoint: run `mdriver` with the `-p` flag, and/or run `driver.pl` with the `-c` flag.

You can get some idea of your program performance and the resulting value you will get for the throughput portion of your grade. **However, the Autolab servers will generate different throughput values, and these will determine your actual score.** This is discussed in more detail in Section 8.

4 Required Functions

Your dynamic storage allocator will implement the following functions, declared in `mm.h` and defined in `mm.c`:

```
bool  mm_init(void);
void *malloc(size_t size);
void  free(void *ptr);
void *realloc(void *ptr, size_t size);
void *calloc(size_t nmemb, size_t size);
bool  mm_checkheap(int);
```

We provide you two versions of memory allocators:

`mm.c`**:** A fully-functional implicit-list allocator. We recommend that you use this code as your starting point.

`mm-naive.c`**:** A functional implementation that runs fast but gets very poor utilization, because it never reuses any blocks of memory.

Your allocator must run correctly on a 64-bit machine. It must support a full 64-bit address space, even though current implementations of x86-64 machines support only a 48-bit address space. The driver `mdriver-emulate` will evaluate your program's correctness using benchmark traces that require the use of a full 64-bit address space.

Your submitted `mm.c` must implement the following functions:

- `mm_init`: Performs any necessary initializations, such as allocating the initial heap area. The return value should be `false` if there was a problem in performing the initialization, `true` otherwise. **You must reinitialize all of your data structures in this function**, because the drivers call your `mm_init` function every time they begin a new trace to reset to an empty heap.

- `malloc`: The `malloc` routine returns a pointer to an allocated block payload of at least `size` bytes. The entire allocated block should lie within the heap region and should not overlap with any other allocated block.

  Your `malloc` implementation must always return 16-byte aligned pointers.

- `free`: The `free` routine frees the block pointed to by `ptr`. It returns nothing. This routine is only guaranteed to work when the passed pointer was returned by an earlier call to `malloc`, `calloc`, or `realloc` and has not yet been freed. `free(NULL)` has no effect.

- `realloc`: The `realloc` routine returns a pointer to an allocated region of at least `size` bytes with the following constraints:

  - if `ptr` is `NULL`, the call is equivalent to `malloc(size)`;
  - if `size` is equal to zero, the call is equivalent to `free(ptr)` and should return `NULL`;
  - if `ptr` is not `NULL`, it must have been returned by an earlier call to `malloc` or `realloc` and not yet have been freed. The call to `realloc` takes an existing block of memory, pointed to by `ptr` — the *old block*. Upon return, the contents of the new block should be the same as those of the old block, up to the minimum of the old and new sizes. Everything else is uninitialized. Achieving this involves either copying the old bytes to a newly allocated region or reusing the existing region.

    For example, if the old block is 16 bytes and the new block is 24 bytes, then the first 16 bytes of the new block are identical to the first 16 bytes of the old block and the last 8 bytes are uninitialized. Similarly, if the old block is 16 bytes and the new block is 8 bytes, then the contents of the new block are identical to the first 8 bytes of the old block.

    The function returns a pointer to the resulting region. The return value might be the same as the old block—perhaps there is free space after the old block, or `size` is smaller than the old block size—or it might be different. If the call to `realloc` does not fail and the returned address is different than the address passed in, the old block has been freed and should not be used, freed, or passed to `realloc` again.

  Hint: Your `realloc` implementation will have only minimal impact on measured throughput or utilization. A correct, simple implementation will suffice.

- `calloc`: Allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory. The memory is set to zero before returning.

  Hint: Your `calloc` will not be graded on throughput or performance. A correct, simple implementation will suffice.

- `mm_checkheap`: The `mm_checkheap` function (the *heap consistency checker*, or simply *heap checker*) scans the heap and checks it for possible errors (e.g., by making sure the headers and footers of each block are identical). Your heap checker should run silently until it detects some error in the heap. Then, and only then, should it print a message and return `false`. If it finds no errors, it should return `true`. It is very important that your heap checker run silently; otherwise, it will produce too much output to be useful on the large traces.

A quality heap checker is essential for debugging your `malloc` implementation. Many `malloc` bugs are too subtle to debug using conventional `gdb` techniques. The only effective technique for some of these bugs is to use a heap consistency checker. When you encounter a bug, you can isolate it with repeated calls to the consistency checker until you find the operation that corrupted your heap. Because of the importance of the consistency checker, it will be graded. If you ask members of the course staff for help, the *first thing we will do is ask to see your checkheap function*, so **you must write this function before coming to see us!** It will save you time debugging.

The `mm_checkheap` function takes a single integer argument that you can use any way you want. One very useful technique is to use this argument to pass in the line number of the call site:

```
mm_checkheap(__LINE__);
```

If `mm_checkheap` detects a problem with the heap, it can print the line number where `mm_checkheap` was called, which allows you to call `mm_checkheap` at numerous places in your code while you are debugging.

Note: `mm_checkheap` is also called with `0` as the argument from the driver.

The semantics for `malloc`, `realloc`, `calloc`, and `free` match those of the corresponding `libc` routines. Type `man malloc` to the shell for complete documentation.

5 Support Routines

The `memlib.c` package simulates the memory system for your dynamic memory allocator. You can invoke the following functions in `memlib.c`:

- `void *mem_sbrk(intptr_t incr)`: Expands the heap by `incr` bytes, where `incr` is a non-negative integer, and returns a generic pointer to the first byte of the newly allocated heap area. The semantics are identical to the Unix `sbrk` function, except that `mem_sbrk` will fail for negative arguments. (Data type `intptr_t` is defined to be a signed integer large enough to hold a pointer. On our machines it is 64-bits long.)
- `void *mem_heap_lo(void)`: Returns a generic pointer to the first byte in the heap.
- `void *mem_heap_hi(void)`: Returns a generic pointer to the last byte in the heap.

  Careful: the definition of `mem_heap_hi()` given here is correct, but it's not necessarily intuitive! If your heap is 16 bytes long, then which byte is the "last byte in the heap"? If you want to get a pointer to the last word in the heap, what do you have to do?
- `size_t mem_heapsize(void)`: Returns the current size of the heap in bytes.

You are also allowed to use the following `libc` library functions: `memcpy`, `memset`, `printf`, `fprintf`, and `sprintf`. Other than these functions and the support routines, your `mm.c` code may not call any externally-defined function.

6 The Trace-driven Driver Programs

Three driver programs generated when you run `make`: `mdriver`, `mdriver-dbg`, and `mdriver-emulate`.

The driver programs test your `mm.c` code for correctness, space utilization, and throughput. These driver programs are controlled by a set of *trace files* that are included in the `traces` subdirectory of the `malloclab-handout.tar` distribution. Each trace file contains a sequence of commands that instruct the driver to call your `malloc`, `realloc`, and `free` routines in some sequence. The drivers and the trace files are the same ones we will use when we grade your handin `mm.c` file.

When the driver programs are run, they will run each trace file multiple times: once to make sure your implementation is correct, once to determine the space utilization, and between 3 and 20 times to determine the throughput.

The drivers accept the following command-line arguments. The normal operation is to run it with no arguments, but you may find it useful to use the arguments during development.

- `-p`: Apply the scoring standards for the checkpoint, rather than for the final submission.
- `-t` *`tracedir`*: Look for the default trace files in directory *`tracedir`* instead of the default directory `traces`
- `-f` *`tracefile`*: Use one particular *`tracefile`* instead of the default set of tracefiles for testing correctness, utilization, and throughput.
- `-c` *`tracefile`*: Run a particular *`tracefile`* twice, calling `mm_init` in between each run, testing only for correctness. This option is extremely useful if you want to print out debugging messages.
- `-h`: Print a summary of the possible command line arguments.
- `-l`: Run and measure the `libc` version of `malloc` in addition to your `malloc` package. This is interesting if you want to see how fast a real `malloc` package runs, but it has no effect on your grade for this assignment.
- `-v` *`level`*: Set the verbosity level to the specified value. Levels 0–2 are supported, with a default level of 1. Raising the verbosity level causes additional diagnostic information to be printed as each trace file is processed. This can be useful during debugging for determining which trace file is causing your `malloc` package to fail.
- `-V`: Equivalent to `-v 2`.
- `-d` *`level`*: At debug level 0, very little validity checking is done. This is useful if you are mostly done but just tweaking performance.

  At debug level 1, every array the driver allocates is filled with random bytes. When the array is freed or reallocated, the driver checks to make sure the bytes have not been changed. This is the default.

  At debug level 2, every time any operation is done, all of the allocated arrays are checked to make sure they still hold their randomly assigned bytes, and your implementation of `mm_checkheap` is called. This mode is slow, but it can help identify the exact point at which an error gets injected.

- `-D`: Equivalent to `-d 2`.
- `-s` s: Time out after s seconds. The default is to never time out.
- `-T`: Print the output in a tabular form. This can be useful if you want to convert the results into a spreadsheet for further analysis.

For most of your testing, you should use the `mdriver` program. It checks the correctness, utilization, and throughput of the standard benchmark traces. The `mdriver-emulate` program uses special compilation and memory-management techniques to allow testing your program with a heap making use of the full, 64-bit address space. In addition to the standard benchmark traces, it will run a set of *giant* traces with very large allocation requests. **If your submission (for either the checkpoint of the final version) fails to pass any of its checks, you will be given a penalty of 30 points.** One of the checks is to verify that any heap location read from has been written to first.

The `mdriver-dbg` is a version of the driver that may be useful for debugging, and is **not** graded by driver.pl or by Autolab. This program will define the `DEBUG` constant, which enables the `dbg_` macros at the top of mm.c. We've provided these macros so that you can cleanly make use of debugging routines or turn them off at your leisure. Additionally, this program is compiled with a lower optimization level `-Og`, which will allow GDB to display more meaningful debugging information.

7 Programming Rules

- You are writing a general purpose allocator. You may not solve specifically for any of the traces—we will be checking for this. Any allocator that attempts to explicitly determine which trace is running (e.g., by executing a sequence of tests at the beginning of the trace) and change its behavior in a way that is appropriate only for that specific trace will receive a penalty of 20 points. On the other hand, you should feel free to write an *adaptive* allocator—one that dynamically tunes itself according to the general characteristics of the different traces.
- You should not change any of the interfaces in `mm.h`, and your program must compile with the provided Makefile. However, we strongly encourage you to use `static` helper functions in `mm.c` to break up your code into small, easy-to-understand segments.
- You are not allowed to define any large global data structures such as large arrays, trees, or lists in your `mm.c` program. However, you *are* allowed to declare small global arrays, structs and scalar variables such as integers, floats, and pointers in `mm.c`. **In total, your global data should sum to at most 128 bytes.** Global values defined with the `const` qualifier are not counted in the 128 bytes.

  The reason for this restriction is that the driver cannot account for such global variables in its memory utilization measure. If you need space for large data structures, you can allocate space for them within the heap.

  The 128-byte limit will be tested by automated means and a penalty will be applied immediately by the driver from testing. Exceeding this limit is permitted as part of your testing and experimentation, but will be applied automatically by the driver to your final Autolab submission.

- The presence of some undefined behavior is inevitable in malloclab, where we perform plenty of unsafe memory operations and generally treat memory as if it is a giant playground. For example, we inevitably need to do things like cast between pointer types, perform pointer arithmetic, and write to arbitrary positions in memory. Outside of this lab, it is rarely appropriate to write code in this style, but it is necessary to do so here.

  However, we do ask you that *minimize* the amount of undefined behavior, and *constrain* it as much as is feasible. For example, instead of directly casting between pointer types, you should explicitly alias memory through the use of unions if possible. Additionally, you should limit the use of pointer arithmetic to a few short helper functions, as we have tried to do in the handout code.

- In the provided baseline code, we use a **zero-length array** to declare a payload element in the block struct. This is a compiler extension specific to gcc (which is also implemented by clang), so its use is not portable, but we encourage you to use it for this lab as a way to explicitly declare your block payload.

  One important restriction is that **a zero-length array must appear as the last element in its containing struct.** However, for malloclab, we will also permit the use of a zero-length array as a member of a union, as long as that union is the last element in its containing struct. This will allow you to explicitly alias the payload with other data that you might also want to store in the same memory address. Though this is not portable, we strongly encourage its use compared to the alternative of casting between pointer types.

- The use of macro definitions (using `#define`) in your code is restricted to the following:

  1. Macros with names beginning with the prefix "`dbg_`" that are used for debugging purposes only. See, for example, the debugging macros defined in `mm.c`. You may create other ones, but they must be disabled in any version of your code submitted to Autolab.
  2. Definitions of constants. These definitions must not have any parameters.

  Explanation: It is traditional in C programming to use macros instead of function definitions in an attempt to improve program performance. This practice is obsolete. Modern compilers (when optimization is enabled) perform *inline substitution* of small functions, eliminating any inefficiencies due to the use of functions rather than macros. In addition, functions provide better type checking and (when optimization is disabled) enable better debugging support.

  Here are some examples of allowed and disallowed macro definitions:

`#define DEBUG 1`	OK	Defines a constant
`#define CHUNKSIZE (1<<12)`	OK	Defines a constant
`#define WSIZE sizeof(uint64_t)`	OK	Defines a constant
`#define dbg_printf(...) printf(__VA_ARGS__)`	OK	Debugging support
`#define GET(p) (*(unsigned int *)(p))`	Not OK	Has parameters
`#define PACK(size, alloc) ((size)|(alloc))`	Not OK	Has parameters

  When you run `make`, it will run a program that checks for disallowed macro definitions in your code. This checker is overly strict—it cannot determine when a macro definition is embedded in a comment or in some part of the code that has been disabled by conditional-compilation directives. Nonetheless, your code must pass this checker without any warning messages.

- The code shown in the textbook (Section 9.9.12, and available from the CS:APP website) is a useful source of inspiration for the lab, but it does not meet the required coding standards. It does not handle 64-bit allocations, it makes extensive use of macros instead of functions, and it relies heavily on low-level pointer arithmetic. Similarly, the code shown in K&R does not satisfy the coding requirements. You should use the provided code `mm.c` as your starting point.

- It is okay to look at any *high-level* descriptions of algorithms found in the textbook or elsewhere, but it is *not* acceptable to copy or look at any code of `malloc` implementations found online or in other sources, except for the allocators described in the textbook, in K&R, or as part of the provided code.

- It is okay to copy code for useful generic data structures and algorithms (e.g., linked lists, hash tables, search trees, and priority queues) from Wikipedia and other repositories. (This code must not have already been customized as part of a memory allocator.) You must include (as a comment) an attribution of the origins of any borrowed code.

- Your allocator must always return pointers that are aligned to 16-byte boundaries. The driver will check this requirement.

- Your code *must* compile without warnings. Warnings often point to subtle errors in your code; whenever you get a warning, you should double-check the corresponding line to see if the code is really doing what you intended. If it is, then you should eliminate the warning by tweaking the code (for instance, one common type of warning can be eliminated by adding a type-cast where a value is being converted from one type of pointer to another). We have added flags in the Makefile to force your code to be error-free. You may remove those flags during development if you wish, but please realize that we will be grading you with those flags activated.

- Your code will be compiled with the `clang` compiler, rather than `gcc`. This compiler is distributed by the LLVM Project (`llvm.org`). Their compiler infrastructure provides features that have enabled us to implement the 64-bit address emulation techniques of `mdriver-emulate`. For the most part, `clang` is compatible with `gcc`, except that it generates different error and warning messages.

8 Evaluation

Malloc Lab is worth 11% of your final grade in the course: 4% for the checkpoint and 7% for the final version. Assuming your solution passes all correctness tests, and the graders do not detect any errors in your source code, two metrics are used to evaluate performance:

- *Space utilization*: The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via `malloc` but not yet freed via `free`) and the size of the heap used by your allocator. The optimal (but unachievable) ratio equals 100%. You should find good policies to minimize fragmentation in order to make this ratio as close as possible to the optimal.

- *Throughput*: The average number of operations completed per second, expressed in *kilo-operations per second* or KOPS. A trace that takes T seconds to perform n operations will have a throughput of $n/(1000 \cdot T)$ KOPS. Throughput measurements vary according to the type of CPU running the

program. We will compensate for this machine dependency by evaluating the throughput of your implementations relative to those of reference implementations running on the same machine.

8.1 Machine Dependencies

You will find that your program gets different throughput values depending on the model of CPU for the machine running the program. Our evaluation code compensates for these differences by comparing the performance of your program to implementations we have written for both the checkpoint and the final versions. It can determine the reference performance for the machine executing the program in two different ways. First, it looks in the file `throughputs.txt` to see if it has a record for the executing CPU. (Linux machines contain a file `/proc/cpuinfo` that includes information about the CPU model.) Second, if it does not find this information, it runs reference implementations that are included as part of the provided files.

Throughput information has been generated for the CPUs in the Shark machines, as well as the three different CPU models used by the Autolab servers, which we refer to as "Autolab A", "Autolab B", and "Autolab C." The three different CPU types are:

Name	ID	Class	Model	Clock (GHz)
Shark	Intel(R)Xeon(R)CPUE5520@2.27GHz	Intel Xeon	E5520	2.27
Autolab A	Intel(R)Xeon(R)CPUE5-2687Wv3@3.10GHz	Intel Xeon	E5-2687	3.10
Autolab B	Intel(R)Xeon(R)CPUE5-2690v2@3.00GHz	Intel Xeon	E5-2690	3.00
Autolab C	Intel(R)Xeon(R)Gold6132CPU@2.60GHz	Intel Xeon	Gold 6132	2.60

When `mdriver` runs, it will print out the CPU model ID (a compressed version of the information in `/proc/cpuinfo`) and the benchmark throughput for that CPU model.

8.2 Performance Points

Observing that both memory and CPU cycles are expensive system resources, we combine these two measures into a single performance index P, with $0 \leq P \leq 100$, computed as a weighted sum of the space utilization and throughput:

$$P\left(U,T\right) = 100\left(w \cdot \mathit{Threshold}\left(\frac{U - U_{min}}{U_{max} - U_{min}}\right) + (1-w) \cdot \mathit{Threshold}\left(\frac{T - T_{min}}{T_{max} - T_{min}}\right)\right)$$

where U is the space utilization (averaged across the traces), and T is the throughput (harmonic mean across the traces). U_{max} and T_{max} are the estimated space utilization and throughput of a well-optimized `malloc` package, and U_{min} are T_{min} are minimum space utilization and throughput values, below which you will receive 0 points. The weight w defines the relative weighting of utilization versus performance in the score. Function $\mathit{Threshold}$ is defined as

$$\mathit{Threshold}(x) \quad = \quad \begin{cases} 0, & x < 0 \\ x, & 0 \leq x \leq 1 \\ 1, & x > 1. \end{cases}$$

The values of T_{min} and T_{max} are computed relative to the performance of reference versions of allocators, with one designed to meet the utilization and throughput goals for the checkpoint, and the other to meet the goals for the final submission. If the reference version provides throughput T_{ref}, then the throughput values used in computing the score are given as:

$$\begin{aligned} T_{min} &= R_{min} \cdot T_{ref} \\ T_{max} &= R_{max} \cdot T_{ref} \end{aligned}$$

where the values of R_{min} and R_{max} differ for the checkpoint and the final versions. The following table shows the evaluation parameters for the checkpoint and final versions

Version	w	U_{min}	U_{max}	R_{min}	R_{max}
Checkpoint	0.2	0.55	0.58	0.1	0.8
Final	0.6	0.55	0.74	0.5	0.9

The following table summarizes the throughput standards, based on CPU model and checkpoint or final version:

	Checkpoint			Final		
Machine	T_{min}	T_{max}	T_{ref}	T_{min}	T_{max}	T_{ref}
Shark	1,211	9,690	12,112	4,004	7,208	8,009
Autolab A	1,231	9,851	12,313	4,029	7,253	8,059
Autolab B	1,363	10,906	13,632	4,538	8,169	9,077
Autolab C	1,319	10,554	13,193	4,630	8,334	9,261

That is, in English, for the checkpoint your score will be computed as 20% utilization and 80% throughput, and for the final it will be 60% utilization and 40% throughput.

Note that you will get an indication of your program's throughput by running it on a Shark machine, but you must submit it to Autolab to get a reliable calibration.

The throughput standards are set low enough that we expect your program will significantly exceed the requirements for T_{max}. Doing so will greatly reduce the frustration of dealing with the different CPU models and the fluctuations from one run to another caused by the system load.

The `mdriver` driver program will assign a performance index of 0 if it detects an error in any of the traces. In addition, we will run a separate test of the driver `mdriver-emulate`. If it detects an error, we will deduct 30 points from P.

The `traces` subdirectory contains a number of traces. Some of them are short traces that do not count toward your memory utilization or throughput but are useful for detecting errors and for debugging. In the driver's output, you will see these marked without a '*' next to them. The traces that count towards both your memory utilization and throughput are marked with a '*' in the output of `mdriver`.

8.3 Assignment Grade

The program `driver.pl` runs the two driver programs and computes the performance index P minus any penalty you receive for failing the tests of `mdriver-emulate`. It is the same program Autolab uses to

evaluate your assignment. Run it with the command-line option `-c` to have it use the evaluation standards for the checkpoint, and without this option for the final submission.

Your score for the checkpoint will be determined by the value generated by running `driver.pl` with command-line option `-c`, plus additional points as follows:

- *Heap Consistency Checker (10 points).* Ten points will be awarded based on the quality of your implementation of `mm_checkheap`. **The heap checker will be graded on your Checkpoint submission.** It is up to your discretion how thorough you want your heap checker to be. The more the checker tests, the more valuable it will be as a debugging tool. However, to receive full credit for this part, we require that you check *all* of the invariants of your data structures. Some examples of what your heap checker should check are provided below. **Specific items will be dependent on your design.**
  - Checking the heap (implicit list, explicit list, segregated list):
    - Check epilogue and prologue blocks.
    - Check each block's address alignment.
    - Check heap boundaries.
    - Check each block's header and footer: size (minimum size), previous/next allocate/free bit consistency, header and footer matching each other.
    - Check coalescing: no two consecutive free blocks in the heap.
  - Checking the free list (explicit list, segregated list):
    - All next/previous pointers are consistent (if A's next pointer points to B, B's previous pointer should point to A).
    - All free list pointers are between `mem_heap_lo()` and `mem_heap_high()`.
    - Count free blocks by iterating through every block and traversing free list by pointers and see if they match.
    - All blocks in each list bucket fall within bucket size range (segregated list).

Your score for the final submission will be determined by the value generated by running `./driver.pl`, plus additional points as follows:

- *Style (10 points).* Your code should follow the Style Guidelines posted on the course Web site. In addition:
  - Your code should be decomposed into functions and use as few global variables as possible. You should use static functions and declared structs and unions to minimize pointer arithmetic and to isolate it to a few places.
  - You should avoid sprinkling your code with numeric constants. Instead, use declarations via `#define` or static constants. Try, as much as possible, to use C data types, and the operators `sizeof` and `offsetof` to define the sizes of various fields and offsets, rather than using fixed numeric values.
  - Your `mm.c` file **must** begin with a header comment that gives an overview of the structure of your free and allocated blocks, the organization of the free list, and how your allocator manipulates the free list.

- In addition to this overview header comment, each function **must** be preceded by a header comment that describes what the function does. Make sure to review the course style guide: we are expecting that for each function, you document at a minimum its purpose, arguments, return value, and any relevant preconditions or postconditions.
- You will want to use inline comments to explain code flow or code that is tricky.
- Your code should be modular, robust, and easily scalable. You should be able to easily change various parameters that define your allocator, without any changes in the actual operation of the program. For example, you should be able to arbitrarily change the number of segregated lists with minimal modifications.
- Your code should avoid undefined behavior when possible. As described in the"Programming Rules" section, though some amount of undefined behavior is inevitable when manipulating memory, you should avoid writing code that unnecessarily invokes undefined behavior, and constrain it as much as is feasible. Furthermore, make sure you keep in mind the restrictions on the use of zero-length arrays described in that section.

Study the code in `mm.c` as an example of the desired coding style.

9 Handin Instructions

Make sure you have included your name and Andrew ID in the header comment of `mm.c`. Make your code does not print anything during normal operation, and that all debugging macros have been disabled.

Hand in your `mm.c` file by uploading it to Autolab. You may submit your solution as many times as you wish until the due date.

Only the last version you submit will be graded.

For this lab, you must upload your code for the results to appear on the class status page.

10 Useful Tips

- We have included the `contracts.h` header file from 15-122 in the handout directory. You'll find debugging macros defined in `mm.c` that provide aliases to the contracts functions. Please feel free to use these to check your invariants during development.
- *Use the drivers'* `-c` *and* `-f` *options.* During initial development, using short trace files will simplify debugging and testing.
- *Use the drivers'* `-V` *option.* The `-V` option will also indicate when each trace file is processed, which will help you isolate errors.
- *Use the drivers'* `-D` *option.* This does a lot of checking to quickly find errors.

- *Use a debugger.* A debugger will help you isolate and identify out-of-bounds memory references. Modify the Makefile to set the parameter `COPT` to `-O0` (big-Oh, numeral zero) to disable optimizations and then recompile your code with the command `make clean ; make`. But, do not forget to restore the Makefile to the original, and then recompile the code, when doing performance testing. This step may cause errors during the build process for `mdriver-emulate`.

- *Use the Clang static analyzer.* The Clang static analyzer is able to find and pinpoint some bugs and provide explanations for them at compile time. To use it, run `make clean` and then run `/usr/local/depot/llvm-7.0/bin/scan-build make` and look for errors that it found in your `mm.c`.

- *Use* `gdb`*'s* `watch` *command* to find out what changed some value you did not expect to have changed.

- *Use the supplied helper function for viewing the heap contents with* `gdb`. You cannot use `gdb` to directly view the contents of the heap when running the version with 64-bit memory emulation. We have provided a helper function for this purpose. See Appendix A for documentation on this function and how to view the heap contents.

- *Reduce obscure pointer arithmetic through the use of* `struct`*'s and* `union`*'s.* Although your data structures will be implemented in compressed form within the heap, you should strive to make them look as conventional as possible using `struct` and `union` declarations to encode the different fields. Examples of this style are shown in the baseline implementation.

- *Encapsulate your pointer operations in functions.* Pointer arithmetic in memory managers is confusing and error-prone because of all the casting that is necessary. You can significantly reduce the complexity by writing functions for your pointer operations. See the code in `mm.c` for examples. **Remember:** you are not allowed to define macros—the code in the textbook does not meet our coding standards for this lab.

- *Your allocator must work for a 64-bit address space*. The `mdriver-emulate` will specifically test this capability. You should allocate a full eight bytes for all of your pointers and size fields. (You *can* take advantage of the low-order bits for some of these values being zero due to the alignment requirements.) Make sure you do not inadvertently invoke 32-bit arithmetic with an expression such as `1<<32`, rather than `1L<<32`.

- *Use your heap consistency checker.* We are assigning ten points to the `mm_checkheap` function in your checkpoint submission for a reason. A good heap consistency checker will save you hours and hours when debugging your `malloc` package. You can use your heap checker to find out where exactly things are going wrong in your implementation (hopefully not in too many places!). Make sure that your heap checker is detailed. To be useful, your heap checker should only produce output when it detects an error. Every time you change your implementation, one of the first things you should do is think about how your `mm_checkheap` will change, what sort of tests need to be performed, and so on. You should have a good implementation of it right from the start. Do not even think about asking for debugging help from any of the course staff until you have implemented and tried using your heap checker.

- *Keep backups.* Whenever you have a working allocator and are considering making changes to it, keep a backup copy of the last working version. It is very common to make changes that inadvertently break the code and then have trouble undoing them.
- *Use version management.* You may find it useful to manage multiple different versions of your implementation (e.g., explicit list, segregated list) as you proceed. Here are three possible approaches:
  1. Use Git. It is recommended that you include current performance results in commit messages.
  2. Use Autolab. Keep submitting different versions to Autolab. You can then retrieve any of them at any point. Your grade will be based only on the final version of the code submitted, and so there is no penalty for submitting multiple, intermediate versions.
  3. Save your different versions under different file names.
- *Start early!* It is possible to write an efficient `malloc` package with a few pages of code. However, we can guarantee that it will be some of the most difficult and sophisticated code you have written so far in your career. So start early, and good luck!

11 Office Hours

This lab has a significant implementation portion that is much more involved than the prior labs. Expect to be debugging issues for several hours - there is no way around this.

Office hours will have a significant wait time as each student has a different implementation and there is not a single "correct" solution.

TAs will **NOT**:

- Go line-by-line through your program to determine where things go wrong.
- Read your code to determine if everything you are doing is correct. There are simply too many subtle issues to go through everyone's programs.

TAs will:

- Help you use GDB efficiently to monitor the state of your program.
- Go through conceptual discussions of what you are doing.

Here are some useful things to have ready before a TA comes to you:

- A non-trivial heap checker that exhaustively tests all the requirements of your implementation.
- Documentation of your data structures (drawings are great!)
- Explicit problems with details about what conditions cause them (not "my malloc doesn't work and I don't know why")

As a reminder, check our office hours schedule on the course website. Start early so you can get help early!

12 Strategic Advice

You must design algorithms and data structures for managing free blocks that achieve the right balance of space utilization and speed. This involves a trade-off—it is easy to write a fast allocator by allocating blocks and never freeing them, or a high-utilization allocator that carefully packs blocks as tightly as possible. You must seek to minimize wasted space while also making your program run fast.

As described in the textbook and the lectures, utilization is reduced below 100% due to *fragmentation*, taking two different forms:

- *External fragmentation:* Unused space between allocated blocks or at the end of the heap
- *Internal fragmentation:* Space within an allocated block that cannot be used for storing data, because it is required for some of the manager's data structures (e.g., headers, footers, and free-list pointers), or because extra bytes must be allocated to meet alignment or minimum block size requirements

To reduce external fragmentation, you will want to implement good block placement heuristics. To reduce internal fragmentation, it helps to reduce the storage for your data structures as much as possible.

Maximizing throughput requires making sure your allocator finds a suitable block quickly. This involves constructing more elaborate data structures than is found in the provided code. However, your code need not use any exotic data structures, such as search trees. Our reference implementation only uses singly- and doubly-linked lists.

Here's a strategy we suggest you follow in meeting the checkpoint and final version requirements:

- *Checkpoint*. The provided code already meets the required utilization performance, but it has very low throughput. You can achieve the required throughput by converting to an explicit-list allocator, and then converting that into a segregated free list allocator. Both of these changes will not affect your utilization much.

  You will want to experiment with allocation policies. The provided code implements first-fit search. Some allocators attempt best-fit search, but this is difficult to do efficiently. You can find ways to introduce elements of best-fit search into a first-fit allocator, while keeping the amount of search bounded.

  Depending on whether you place newly freed blocks at the beginning or the end of a free list, you can implement either a last-in-first-out (LIFO) or a first-in-first-out (FIFO) queue discipline. You should experiment with both.

- *Final Version*. Building on the checkpoint version, you must greatly increase the utilization and keep a high throughput. You must reduce *both* external and internal fragmentation. Reducing external fragmentation requires achieving something closer to best-fit allocation. Reducing internal fragmentation requires reducing data structure overhead. There are multiple ways to do this, each with its own challenges. Possible approaches and their associated challenges include:

  – Eliminate footers in allocated blocks. But, you still need to be able to implement coalescing. See the discussion about this optimization on page 852 of the textbook.

- Decrease the minimum block size. But, you must then manage free blocks that are too small to hold the pointers for a doubly linked free list.
- Reduce headers below 8 bytes. But, you must support all possible block sizes, and so you must then be able to handle blocks with sizes that are too large to encode in the header.
- Set up special regions of memory for small, fixed-size blocks. But, you will need to manage these and be able to free a block when given only the starting address of its payload.

See Appendix A.3 for a table on approximate expected quantitative results from these optimizations.

Some advice on how to implement and debug your packages will be covered in the lectures and recitations, as well as the Malloc Lab Bootcamp.

Good luck!

A Viewing Heap Contents with GDB

The debugging program `gdb` can be a valuable tool for tracking down bugs in your memory allocator. We hope by this point in the course that you are familiar with many of the features of `gdb`. You will want to take full advantage of them.

Unfortunately, the normal `gdb` commands cannot be used to examine the heap when running `mdriver-emulate`. In this appendix, we present a brief tutorial on using `gdb` with your program and describe a helper function that can be used to examine the heap with `gdb` for both `mdriver` and `mdriver-emulate`. In this tutorial, we use the code in `mm.c` as the reference implementation.

A.1 Viewing the heap without a helper function

A typical `gdb` session to examine the header of a block on a call to free might go something like this. In the following, all text in italics was typed by the user. The session has been edited to remove some uninteresting parts of the printout.

```
linux>  gdb mdriver
(gdb)  break mm_free
Breakpoint 1 at 0x4043a1: file mm.c, line 288.
(gdb)  run -c traces/syn-array-short.rep
Breakpoint 1, mm_free (bp=bp@entry=0x80000eac0) at mm.c:288
(gdb)  print bp
$1 = (void *) 0x80000eac0
(gdb)  print /x *((unsigned long *) bp - 1)
$2 = 0x41a1
(gdb)  quit
```

A few things about this session are worth noting:

- The function named "`free`" in `mm.c` is known to `gdb` in its *unaliased* form as "`mm_free`." You can see that the aliasing is introducted through a macro definition at the beginning of the file. When you use `gdb`, you refer to the unaliased function names. The unaliased names of other important functions in `mm.c` include: `mm_malloc`, `mm_realloc`, `mm_calloc`, `mem_memset`, and `mem_memcpy`.
- The `gdb` command "`print /x *((unsigned long *) bp - 1)`" first casts the argument to `free` to be a pointer to an `unsigned long`. It then decrements this pointer to point to the block header and then prints it in hex format.
- The printed value `0x41a1` indicates that the block is of size `0x41a0` (decimal 16,800), and the lower-order bit is set to indicate that the block is allocated. Looking at the trace file, you will see that the block to be freed has a payload of 16,784 bytes. This required allocating a block of size 16,800 to hold the header, payload, and footer.

When we try the same method with `mdriver-emulate`, things don't work as well. In this case, we use one of the traces with giant allocations, but the same problem will be encountered with any of the traces.

```
linux>  gdb mdriver-emulate
(gdb)  break mm_free
Breakpoint 1 at 0x4043b7: file mm.c, line 285.
(gdb)  run -c traces/syn-giantarray-short.rep
Breakpoint 1, mm_free (bp=bp@entry=0x23368bd380eb2cb0) at mm.c:285
(gdb)  print bp
$1 = (void *) 0x23368bd380eb2cb0
(gdb)  print /x *((unsigned long *) bp - 1)
Cannot access memory at address 0x23368bd380eb2ca8
(gdb)  quit
```

The problem is that `bp` is set to `0x23368bd380eb2cb0`, or around 2.54×10^{18}, which is well beyond the range of virtual addresses supported by the machine. The `mdriver-emulate` program uses sparse memory techniques to provide the illusion of a full, 64-bit address space, and so it supports very large pointer values. However, the actual addressing has an overall limit of 100 MB of actual memory usage.

A.2 Viewing the heap with the `hprobe` helper function

To support the use of `gdb` in debugging both the normal and the emulated version of the memory, we have created a function with the following prototype:

```
void hprobe(void *ptr, int offset, size_t count);
```

This function will print the `count` bytes that start at the address given by summing `ptr` and `offset`. Having a separate offset argument eliminates the need for doing pointer arithmetic in your queury.

Here's an example of using `hprobe` with `mdriver`:

```
linux>  gdb mdriver
(gdb)  break mm_free
Breakpoint 1 at 0x4043a1: file mm.c, line 288.
(gdb)  run -c traces/syn-array-short.rep
Breakpoint 1, mm_free (bp=bp@entry=0x80000eac0) at mm.c:288
(gdb)  print bp
$1 = (void *) 0x80000eac0
(gdb)  print hprobe(bp, -8, 8)
Bytes 0x80000eabf...0x80000eab8: 0x00000000000041a1
(gdb)  quit
```

Observe that `hprobe` is called with the argument to `free` as the pointer, an offset of -8 and a count of 8. The function prints the bytes with the most significant byte on the left, just as for normal printing of numeric

values. The range of addresses is shown as *HighAddr*`...`*LowAddr*. We can see that the printed value is identical to what was printed using pointer arithmetic, but with leading zeros added.

The same command sequence works for `mdriver-emulate`:

```
linux>  gdb mdriver-emulate
(gdb) break mm_free
Breakpoint 1 at 0x4043b7: file mm.c, line 285.
(gdb)  run -c traces/syn-giantarray-short.rep
Breakpoint 1, mm_free (bp=bp@entry=0x23368bd380eb2cb0) at mm.c:285
(gdb)  print bp
$1 = (void *) 0x23368bd380eb2cb0
(gdb)  print hprobe(bp, -8, 8)
Bytes 0x23368bd380eb2caf...0x23368bd380eb2ca8: 0x00eb55b00c8f1ed1
(gdb)  quit
```

The contents of the header indicate a block size of `0xeb55b00c8f1ed0` (decimal 66,240,834,140,315,344), with the low-order bit set to indicate that the block is allocated. Looking at the trace, we see that the block to be freed has a payload of 66,240,834,160,315,328 bytes. Sixteen additional bytes were required for the header and the footer.

Part of being a productive programmer is to make use of the tools available. Many novice programmers fill their code with print statements. While that can be a useful approach, it is often more efficient to use debuggers, such as `gdb`. With the `hprobe` helper function, you can use `gdb` on both versions of the driver program.

A.3 Approximate Expected Results from Optimizations

Optimization	Utilization	Throughput
Implicit List (Starter Code)	59%	10-100
Explicit Free List	-	2000-5000
Segregated Free Lists	-	11000
Better Fit Algorithm	59%	Variable
Eliminating Footers in Allocated Blocks	+9%	-
Decreasing Block Size/Mini Blocks	+6%	-20%
Compressing Headers	+2%	-

- indicates no change.